Linear-time Self-stabilizing Byzantine Clock Synchronization
(updated version)

Ariel Daliot[1], Danny Dolev[1], and Hanna Parnas[2]

[1] School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel.
{adaliot,dolev}@cs.huji.ac.il
[2] Department of Neurobiology and the Otto Loewi Minerva Center for Cellular and Molecular Neurobiology, Institute of Life Science, The Hebrew University of Jerusalem, Israel.
hanna@vms.huji.ac.il

Abstract. Clock synchronization is a very fundamental task in distributed system. The vast majority of distributed tasks require some sort of synchronization and clock synchronization is a very straightforward tool for supplying this. It thus makes sense to require an underlying clock synchronization mechanism to be highly fault-tolerant. A self-stabilizing algorithm seeks to attain synchronization once lost; a Byzantine algorithm assumes synchronization is never lost and focuses on containing the influence of the permanent presence of faulty nodes. There are efficient self-stabilizing solutions for clock synchronization as well as efficient solutions that are resilient to Byzantine faults. In contrast, to the best of our knowledge there is no practical solution that is self-stabilizing while tolerating the permanent presence of Byzantine nodes. Designing algorithms that self-stabilize while at the same time tolerate permanent Byzantine failures present a special challenge due to the "ambition" of malicious nodes to hamper stabilization if the system tries to recover from a corrupted state. We present the first linear-time self-stabilizing Byzantine clock synchronization algorithm. Our deterministic clock synchronization algorithm is based on the observation that all clock synchronization algorithms require events for exchanging clock values and re-synchronizing the clocks to within safe bounds. These events usually need to happen synchronously at the different nodes. In classic Byzantine algorithms this is fulfilled or aided by having the clocks initially close to each other and thus the actual clock values can be used for synchronizing the events. This implies that clock values cannot differ arbitrarily, which necessarily renders these solutions to be non-stabilizing. Our scheme suggests using an underlying distributed pulse synchronization module that is uncorrelated to the clock values. The synchronized pulses are used as the events for re-synchronizing the clock values. The algorithm is very efficient and attains and maintains high precision of the clocks.

This is an updated version. The original paper appeared in OPODIS'03. The main difference is the replacement of the pulse synchronization module.

1 Introduction

On-going faults whose nature is not predictable or that express complex behavior are most suitably addressed in the Byzantine fault model. It is the preferred fault

model in order to seal off unexpected behavior within limitations on the number of concurrent faults. Most distributed tasks require the number of concurrent Byzantine faults, f, to abide by the ratio of $3f < n$, where n is the network size. See [14] for impossibility results on several consensus related problems such as clock synchronization. Additionally, it makes sense to require systems to resume operation after a major failure without the need for an outside intervention and/or a restart of the system from scratch. E.g. systems may occasionally experience short periods in which more than a third of the nodes are faulty or messages sent by all nodes may be lost for some time due to a network failure.

Such transient violations of the basic fault assumptions may leave the system in an arbitrary state from which the protocol is required to resume in realizing its task. Typically, Byzantine algorithms do not ensure convergence in such cases, as strong assumptions are usually made on the initial state and thus merely focus on preventing Byzantine faults from notably shifting the system state away from the goal. A *self-stabilizing* algorithm bypasses this limitation by being designed to converge within finite time to a desired state from any initial state. Thus, even if the system loses its consistency due to a transient violation of the basic fault assumptions (e.g. more than a third of the nodes being faulty, network disconnected, etc.), then once the system becomes coherent again the protocol will successfully realize the task, irrespective of the resumed state of the system. In trying to combine both fault models, Byzantine failures present a special challenge for designing stabilizing algorithms due to the "ambition" of malicious nodes to incessantly hamper stabilization, as might be indicated by the remarkably few algorithms resilient to both fault models. For a short survey of self-stabilization see [3], for an extensive study see [11].

The current paper addresses the problem of synchronizing clocks in a distributed system. There are several efficient algorithms for self-stabilizing clock synchronization withstanding crash faults (see [13,18,10], for other variants of the problem see [2,15]). There are many efficient classic Byzantine clock synchronization algorithms, for a performance evaluation of clock synchronization algorithms see [1]. However, strong assumptions on the initial state of the nodes are typically made, usually assuming all clocks are initially synchronized ([1,8,21]) and thus these are not self-stabilizing solutions. On the other hand, self-stabilizing clock synchronization algorithms allow initialization with arbitrary clock values, but typically have a cost in the convergence times or in the severity of the faults contained. Evidently, there are very few self-stabilizing solutions facing Byzantine faults ([12]), all with unpractical convergence times. The protocols in [12] are to the best of our knowledge the first self-stabilizing protocols that are tolerant to Byzantine faults. Note that self-stabilizing clock synchronization has an inherent difficulty in estimating real-time without an external time reference due to the fact that non-faulty nodes may initialize with arbitrary clock values. Thus, self-stabilizing clock synchronization

aims at reaching a legal state from which clocks proceed synchronously at the rate of real-time (assuming that nodes have access to physical timers which rate is close to real-time) and not necessarily at estimating real-time. Many applications utilizing the synchronization of clocks do not really require the exact real-time notion (see [16]). In such applications, agreeing on a common clock reading is sufficient as long as the clocks progress within a linear envelope of any real-time interval.

We present a Byzantine self-stabilizing clock synchronization protocol with the following property: should the system initialize or recover from any transient faults with arbitrary clock values then the clocks of the correct nodes proceed synchronously at real-time rate. Should the clocks of the correct nodes hold values that are close to real-time, then the correct clocks proceed synchronously with high real-time accuracy. Thus, the protocol we present significantly improves upon existing Byzantine self-stabilizing clock synchronization algorithms by reducing the time complexity from expected exponential ([12]) to deterministic $O(f)$. Our protocol improves upon existing Byzantine non-stabilizing clock synchronization algorithms by providing self-stabilization while performing with similar complexity. The self-stabilization and comparably low complexity is achieved by executing on top of a deterministic Byzantine self-stabilizing algorithm for pulse synchronization [5]. The interval between the synchronized pulses is long enough to allow initialization and termination of a Byzantine consensus procedure on the clock values, thus attaining and maintaining a common clock reading.

Having access to an outside source of real-time is useful. In such case our approach maintains a consistent system state when the outside source fails.

A special challenge in self-stabilizing clock synchronization is the clock wrap around. In non-stabilizing algorithms having a large enough integer eliminates the problem for any practical concern. In self-stabilizing schemes a transient failure can cause clocks to hold arbitrary large values, surfacing the issue of clock bounds. Our clock synchronization scheme handles clock wrap around difficulties.

The system may be in an arbitrary state in which the communication network may behave arbitrarily and in which there may be an unbounded number of concurrent Byzantine faulty nodes. The algorithm will eventually converge once the communication network resumes delivering messages within bounded, some d, time units, and the fraction of Byzantine nodes, f, obeys $n \geq 3f + 1$, for a network of size n. The attained clock precision and accuracy is $11d$ real-time units, though we present an additional scheme that can attain clock precision and accuracy of $3d$. The convergence time is $O(f')$ communication rounds, where $f' \leq f$ is the actual number of concurrent faults. Our protocol has the additional advantage of a minimal time and message overhead during steady-state after the clocks have synchronized.

An additional advantage of our algorithm is the use of a Byzantine Consensus protocol that works in a message driven manner. The basic protocol follows closely

the early stopping Byzantine Agreement protocol of Toueg, Perry and Srikanth [20]. The main difference is that the protocol rounds progress at the rate of the actual time of information exchange among the correctly operating nodes. This, typically, is much faster than progression with rounds whose time lengths are functions of the upper bound on message delivery time between correct nodes.

2 Model and Problem Definition

The environment is a bounded-delay network model of n nodes that communicate by exchanging messages. We assume that the message passing allows for an authenticated identity of the senders. The communication network does not guarantee any order on messages among different nodes. Individual nodes have no access to a central clock and there is no external pulse system. The hardware clock rate (referred to as the *physical timers*) of correct nodes has a bounded drift, ρ, from real-time rate. Consequent to transient failures there can be an arbitrary number of concurrent Byzantine faulty nodes, the turnover rate between faulty and non-faulty behavior of the nodes can be arbitrary and the communication network may behave arbitrarily. Eventually the system behaves coherently again but in an arbitrary state.

Definition 1. *A node is* **non-faulty** *at times that it complies with the following:*

1. *Obeys a global constant* $0 < \rho << 1$ *(typically* $\rho \approx 10^{-6}$*), such that for every real-time interval* $[u, v]$:

$$(1-\rho)(v-u) \leq \text{ 'physical timer'}(v) - \text{ 'physical timer'}(u) \leq (1+\rho)(v-u).$$

2. *Operates according to the instructed protocol.*
3. *Processes any message of the instructed protocol within* π *real-time units of arrival time.*

A node is considered **faulty** if it violates any of the above conditions. We allow for Byzantine behavior of the faulty nodes. A faulty node may recover from its faulty behavior once it resumes obeying the conditions of a non-faulty node. For consistency reasons, the "correction" is not immediate but rather takes a certain amount of time during which the non-faulty node is still not counted as a correct node, although it supposedly behaves "correctly"[1]. We later specify the time-length of continuous non-faulty behavior required of a recovering node to be considered **correct**.

Definition 2. *The communication network is* **non-faulty** *at periods that it complies with the following:*

[1] For example, a node may recover with arbitrary variables, which may violate the validity condition if considered correct prematurely.

1. *Any message sent by any non-faulty node arrives at every non-faulty node within δ real-time units;*
2. *All messages sent by a non-faulty node and received by a non-faulty node obey FOFI order.*

The system is said to be coherent only following some minimal[2] amount of time of continuous non-faulty behavior of the nodes and the communication network.

Basic notations:

We use the following notations though nodes do not need to maintain all of them as variables.

- $d \equiv \delta + \pi$. Thus, when the communication network is non-faulty, d is the upper bound on the elapsed real-time from the sending of a message by a non-faulty node until it is received and processed by every correct node.
- $Clock_i$, the clock of node i, is a real value in the range 0 to $M-1$. Thus $M-1$ is the maximal value a clock can hold. Its progression rate is a function of node p_i's physical timer. The clock is incremented every time unit. $Clock_i(t)$ denotes the value of the clock of node p_i at real-time t.
- γ is the target upper bound on the difference of clock readings of any two correct clocks at any real-time. Our protocol achieves $\gamma = 3d + O(\rho)$.
- Let $a, b, g, h \in R^+$ be constants that define the linear envelope bound of the correct clock progression rate during any real-time interval.
- $\Psi_i(t_1, t_2)$ is the amount of clock time elapsed on the clock of node p_i during a real-time interval $[t_1, t_2]$ within which p_i was continuously correct. The value of Ψ is not affected by any wrap around of $clock_i$ during that period.
- A ***pulse*** is an internal event targeted to happen in tight synchrony at all correct nodes. A ***Cycle*** (with upper-case initial letter) is the "ideal" time interval length between two successive pulses that a node invokes, as given by the user. The actual cycle length, denoted with lowercase initial, has upper and lower bounds as a result of faulty nodes and the physical clock skew, denoted $cycle_{max}$ and $cycle_{min}$ respectively.
- σ represents the upper bound on the real-time between the invocation of the pulses of different correct nodes (*tightness of pulse synchronization*). The pulse synchronization procedure in [5] achieves $\sigma = 3d$.
- *pulse_conv* represents the convergence time of the underlying pulse synchronization module. The pulse procedure in [5] converges within $6 \cdot cycle$.
- *agreement_duration* represents the maximum real-time required to complete the chosen Byzantine consensus procedure used in Section 3.1. We assume

[2] An infinitely small time period in which the nodes and the communication network are non-faulty has no practical meaning. The required minimal value in our context will be specified later.

$\sigma \leq \sigma + agreement\_duration < cycle \leq Cycle + agreement\_duration$. For simplicity of our arguments we also assume that $M > agreement\_duration$ but this is not a necessary assumption.

Non-faulty nodes do not initialize with arbitrary values of n, f and $Cycle$ as these are fixed constants. It is required that $Cycle$ is chosen s.t. $cycle_{min}$ is large enough to allow our protocol to terminate in between pulses.

A recovering node should be considered correct only once it has been continuously non-faulty for enough time to enable it to go through a complete "synchronization process". This is the time it takes, from any state, to complete a pulses that is in synchrony with all other correct nodes and synchronize with the consensus variables.

Definition 3. *The communication network is* **correct** *following Δ_{net} real-time of continuous non-faulty behavior.*[3]

Definition 4. *A node is* **correct** *following Δ_{node} real-time of continuous non-faulty behavior during a period that the communication network is correct.*[4]

Definition 5. *The system is said to be* **coherent** *at times that it complies with the following:*

1. (Quorum) *At least $n - f$ of the nodes are correct, where $n \geq 3f + 1$;*
2. (Network Correctness) *The communication network is correct.*

Hence, if the system is not coherent then there can be an unbounded number of concurrent faulty nodes; the turnover rate between faulty and non-faulty nodes can be arbitrarily large and the communication network may behave arbitrarily. When the system is coherent, then the communication network and a large enough fraction of the nodes ($n - f$) have been non-faulty for a sufficiently long time period for the pre-conditions for convergence of the protocol to hold. The assumption in this paper, as underlies any other self-stabilizing algorithm, is that eventually the system becomes coherent.

Basic definitions:

- The **clock_state** of the system at real-time t is given by:

$$clock\_state(t) \equiv (clock_0(t),\ \ldots,\ clock_{n-1}(t)) \ .$$

- The systems is in a **synchronized clock_state** at real-time t if $\forall$ *correct* p_i, p_j,

$$(|clock_i(t) - clock_j(t)| \leq \gamma) \ \vee \ (|clock_i(t) - clock_j(t)| \geq M - \gamma) \ .^{5}$$

[3] We will use $\Delta_{net} \geq pulse\_conv + agreement\_duration + \sigma$.
[4] We will use $\Delta_{node} \geq pulse\_conv + agreement\_duration + \sigma$.
[5] The second condition is a result of dealing with bounded clock variables.

Definition 6. The "Self-stabilizing Byzantine Clock Synchronization Problem"

Convergence: *Starting from an arbitrary system state, s, the system reaches a synchronized clock_ state after a finite time.*

Closure: *If s is a synchronized clock_ state of the system at real-time t_0 then $\forall$ real time $t \geq t_0$,*

1. *clock_ state(t) is a synchronized clock_ state,*
2. *"Linear Envelope": for every correct node, p_i,*

$$a \cdot [t - t_0] + b \leq \Psi_i(t_0, t) \leq g \cdot [t - t_0] + h \ .$$

The second Closure condition intends to bound the effective clock progression rate in order to defy a trivial solution.

3 Self-stabilizing Byzantine Clock Synchronization

A major challenge of self-stabilizing clock synchronization is to ensure clock synchronization even when nodes may initialize with arbitrary clock values. This, as mentioned before, requires handling the wrap around of clock values. The algorithm we present employs as a building block an underlying self-stabilizing Byzantine pulse synchronization procedure presented in [5]. In the pulse synchronization problem nodes invoke pulses regularly, ideally every *Cycle* time units. The goal is for the different correct nodes to do so in tight synchrony of each other. To synchronize their clocks, nodes execute at every pulse Byzantine consensus on the clock value to be associated with the next pulse event[6]. When pulses are synchronized, then the consensus results in synchronized clocks. The basic algorithm uses strong consensus to ensure that once correct clocks are synchronized at a certain pulse, and thus enter the consensus procedure with identical values, then they terminate with the same identical values and keep the progression of clocks continuous and synchronized[7].

3.1 The Basic Clock Synchronization Algorithm

The basic clock synchronization algorithm is essentially a self-stabilizing version of the Byzantine clock synchronization algorithm in [8].

We call it PBSS-Clock-Synch (for *Pulse-based Byzantine Self-stabilizing Clock Synchronization*). The agreed clock time to be associated with the next pulse (next

[6] It is assumed that the time between successive pulses is sufficient for a Byzantine consensus algorithm to initiate and terminate in between.

[7] The pulse synchronization building block does not use the value of the clock to determine its progress, but rather intervals measured on the physical timer.

"time for synchronization" in [8]) is denoted by ET (for *Expected Time*, as in [8]). Synchronization of clocks is targeted to happen every *Cycle* time units, unless the pulse is invoked earlier (or later)[8].

```
Algorithm PBSS-Clock-Synch
at "pulse" event                                   /* received the internal pulse event */
begin
 1. Clock := ET;
 2. Revoke possible other instances of PBSS-Clock-Synch and
        clear all data structures besides ET and Clock;
 3. Wait until σ(1 + ρ) time units have elapsed since pulse;
 4. Next_ET := Byz_Consensus((ET + Cycle) mod M, σ);
 5. Clock := (Clock + Next_ET − (ET + Cycle)) mod M;        /* posterior adjust. */
 6. ET := Next_ET;
end
```

Fig. 1. The self-stabilizing Byzantine clock synchronization algorithm

The internal pulse event is delivered by the pulse synchronization procedure. We assume the use of the pulse synchronization presented in [5], though any pulse synchronization algorithm that delivers synchronized pulses by solving the "Self-stabilizing Pulse Synchronization Problem", in the presence of at most f Byzantine nodes, where $n \geq 3f + 1$, such as the pulse procedure in [4], can be executed in the background.

The pulse event aborts any possible on-going invocation of PBSS-Clock-Synch (and thus any on-going instant of Byz_Consensus) and resets all buffers. The synchronization of the pulses ensures that the PBSS-Clock-Synch procedure is invoked within σ real-time units of its invocation at all other correct nodes.

Line 1 sets the local clock to the pre-agreed time associated with the current pulse event. Line 3 intends to make sure that all correct nodes invoke Byz_Consensus only after the pulse has been invoked at all others, without remnants of past invocations, which are revoked at Line 2. Past remnants may exist only during or immediately following periods in which the system is not coherent.

In Line 4 Byz_Consensus intends to reach consensus on the next value of ET. One can use a synchronous consensus algorithm with rounds of size $(\sigma + d)(1 + 2\rho)$ or asynchronous style consensus in which a node waits to get $n - f$ messages of the previous round before moving to the next round. We assume the use of a Byzantine consensus procedure tolerating f faults when $n \geq 3f + 1$. A correct node joins

[8] *Cycle* has the same function as PER in [8].

Byz_Consensus only concomitant to an internal pulse event, as instructed by the PBSS-Clock-Synch. This contains the possibility of faulty nodes to initiate consensus at arbitrary times.

Line 5 is a posterior clock adjustment. It increments the clock value with the difference between the agreed time associated with the next pulse and the node's pre-consensus estimate for the time associated with the next pulse (the value which it entered the consensus with). This is equivalent to incrementing the value of ET that the node was supposed to hold at the pulse according to the agreed $Next\_ET$ with the elapsed time from the pulse and until the termination of Byz_Consensus. This intends to expedite the time to reach synchronization of the clocks. In case that the clock_state before Line 5 was not a synchronized clock_state then a synchronized clock_state is attained following termination of Byz_Consensus at all correct nodes, rather than at the next pulse event. Note that in the case that all correct nodes hold the same ET value at the pulse, then the posterior clock adjustment adds a zero increment to the clock value.

Note that when the system is not yet coherent, following a chaotic state, pulses may arrive to different nodes at arbitrary times, and the ET values and the clocks of different nodes may differ arbitrarily. At that time not all correct nodes will join Byz_Consensus and no consistent resultant value can be guaranteed. Once the pulses synchronize (guaranteed by the pulse synchronization procedure to happen within a single cycle) all correct nodes will join the same instant of Byz_Consensus and will agree on the clock value associated with the next pulse. From that time on, as long as the system stays coherent the clock_state remains a synchronized clock_state.

The use of Byzantine consensus tackles the clock wrap-around in a trivial manner at all correct nodes.

Note that instead of simply setting the clock value to ET we could use some *Clock-Adjustment* procedure (cf. [8]), which receives a parameter indicating the target value of the clock. The procedure runs in the background, it speeds up or slows down the clock rate to smoothly reach the adjusted value within a specified period of time. This procedure should also handle the clock wrap around.

Theorem 1. PBSS-Clock-Synch *solves the "Self-stabilizing Byzantine Clock Synchronization Problem".*

Proof. **Convergence:** Let the system be coherent but in an arbitrary state s, with the nodes holding arbitrary clock values. Consider the first correct node that completed line 3 of the PBSS-Clock-Synch algorithm. Since the system is coherent, all correct nodes invoked the preceding *pulse* within σ of each other. At the last pulse all remnants of previously invoked instances of Byz_Consensus were flushed by all the correct nodes. A correct node does not initiate or join procedure Byz_Consensus

before waiting $\sigma(1+\rho)$ time units subsequent to the pulse, hence not before all correct nodes have invoked a pulse and subsequently flushed their buffers. Thus all correct nodes will eventually join Byz_Consensus, thus Byz_Consensus will initiate and terminate successfully.

At termination of the first instance of Byz_Consensus following the synchronization of the pulses, all correct nodes agree on the clock value to be associated with the next pulse invocation. Subsequently, all correct nodes adjust their clocks, post factum, according to the agreed ET. Note that this posterior adjustment of the clocks does not affect the time span until the invocation of the next pulse but rather updates the clocks concomitantly to and in accordance with the newly agreed ET. This has an effect only if the correct nodes joined Byz_Consensus with differing values. Hence if all correct nodes join Byz_Consensus with the same ET then the adjustment equals zero. Since all correct pulses arrived within σ real-time units of each other, after the posterior clock adjustment of the last correct node, all correct clocks values are within

$$\gamma_1 = \sigma(1+\rho) + (\sigma + \mathit{agreement\_duration}) \cdot 2\rho$$

of each other. The 2ρ is the maximal drift rate between any two correct clocks (whereas ρ is their drift with respect to real-time). Observe that $\gamma_1 \leq \gamma$ and therefore the state of the system is a synchronized clock_state. This concludes the Convergence condition.

□

Closure: Recall that system coherence is defined as a continuous non-faulty behavior of the communication network and a large enough fraction of the nodes for at least some minimal period of time. The proof of the Closure condition assumes the correct nodes have synchronized their ET values, thus setting this minimal time to be at least $\mathit{cycle}_{max} + \mathit{agreement\_duration}$ time, ensuring synchronization of the variables.

Let the system be in a synchronized clock_state and w.l.o.g. assume all correct nodes hold synchronized and identical ET values. Observe that although the correct nodes have synchronized their ET values this does not necessarily imply all correct nodes hold the same ET value at every point in time. At a brief time subsequent to the termination of Byz_Consensus, only a part of the correct nodes may have set the ET to the new agreed value while the rest of the correct nodes currently holding the old ET value will set ET to the new value in a brief time. We first prove the first Closure condition (*precision*). In this case, each correct node adjusts its clock immediately subsequent to the pulse, but the posterior clock adjustment has no effect since the consensus value equals the value it joined Byz_Consensus with. To simplify the discussion assume for now that no wrap around of any correct clock takes place during the time that the pulse arrives at the first correct node and until the pulse is invoked at the last correct node. Immediately after the pulse is invoked

at the last correct node and its subsequent clock adjustment, all correct clocks are within $\gamma_0 = \sigma(1+\rho)$ of each other.

From that point on, clocks of correct nodes drift apart at a rate of 2ρ of each other. As long as no wrap around of the clocks takes place and no pulse arrives at any correct node, the clocks are at most $\gamma_0 + \Delta T \cdot 2\rho$ apart, where ΔT is the real-time elapsed since the invocation of the pulse at the first correct node. To estimate the maximal clock difference, γ, at any time, we will consider the following complementary cases:

P1) Prior to the next pulse event at the first correct node.
P2) When a pulse arrives at some correct node.
P3) Immediately after the last node invokes its next pulse event.

Note that in this case we do not need to consider the posterior adjustment of the clocks at Line 5.

Case P1 cannot last more than $\Delta T = cycle_{max}$, since by the end of that time interval all correct nodes will have invoked the pulse, reducing to case P2 or P3. The discussion above implies $\gamma = \gamma_0 + cycle_{max} \cdot 2\rho$.

Case P3 implies that clock readings are at most γ_0 apart, since all nodes invoke the pulses within σ.

To analyze case P2 consider that the next pulse event has been invoked at some node, p. The following situations may take place:

P2a) Following its clock adjustment, the clock of p holds the maximal clock value among all correct clocks at that moment.
P2b) Following its clock adjustment, the clock of p holds the minimal clock value among all correct clocks at that moment.
P2c) Neither of the above.

In case P2a, since p holds the maximal clock value, we claim that no other clock reading can read less than $ET_{lastpulse} + cycle_{min} \cdot (1-\rho)$. Assume by contradiction the existence of a correct node q whose clock reading is less than this value. Further assume that node q received the same set of messages from the same sources and at the same time as node p. These events caused node p to invoke its pulse and would necessarily cause node q to also invoke a pulse. The elapsed time on the clock of node q between the current pulse and the previous is thus less than $cycle_{min} \cdot (1-\rho)$ which is less than $cycle_{min}$ real-time after its previous pulse. A contradiction to the definition of $cycle_{min}$. Node p just adjusted its clock which thus reads $ET = ET_{lastpulse} + Cycle$. Due to the clock skew the clock difference may increase an additional $2\rho\sigma$ until the node invokes its pulse and the case reduces to P3. The discussion above implies $\gamma = (ET_{lastpulse} + Cycle) - (ET_{lastpulse} + cycle_{min} \cdot (1-\rho)) + 2\rho\sigma = Cycle - cycle_{min} \cdot (1-\rho) + 2\rho\sigma$.

In case P2b, the clock readings of all other nodes that have invoked a pulse can not be more than γ_0 apart (case P3). The clock reading of any node that has not

invoked a pulse yet should be less than $cycle_{max}$ following similar reasoning as in case P2a. Node p just adjusted its clock which thus reads $ET = ET_{lastpulse} + Cycle$. Due to the clock skew the clock difference may increase an additional $2\rho\sigma$ until the node invokes its pulse and the case reduces to P3. The discussion above implies $\gamma = (ET_{lastpulse} + cycle_{max} \cdot (1+\rho)) - (ET_{lastpulse} + Cycle) + 2\rho\sigma = cycle_{max} \cdot (1+\rho) - Cycle + 2\rho\sigma$.

For case P2c, if the nodes holding the minimal clock reading and maximal clock reading already invoked pulses, then the clock difference reduces to case P3.

If neither of the nodes holding the minimal and maximal clock values have not invoked their pulses yet, then the clock difference reduces to case P1.

Otherwise, if either the node holding the minimal or the maximal clock value already invoked its pulse then one of the bounds of P2a or P2b hold until the other node invokes its pulse.

We now consider the case that a clock wrap around takes place at some ΔT real-time after the last pulse is invoked in the synchronized cycle. From the discussion earlier we learn that at the moment prior to the first correct clock wraps around, the correct clocks are at most γ apart. Therefore, all correct clocks will wrap around within at most another γ time. During the intermediate time, any two correct clocks, i, j, for which one has wrapped around and the other not, satisfy $|clock_i(t) - clock_j(t)| \geq M - \gamma$. Thus we proved that the maximal clock difference will remain less than γ or greater than $M - \gamma$, which completes the first Closure condition.

Henceforth, the bound on the clock differences of correct nodes will equal the maximal of the three values calculated above. Formally this yields $\gamma = max[cycle_{max} \cdot (1+\rho) - Cycle + 2\rho\sigma,\ Cycle - cycle_{min} \cdot (1-\rho) + 2\rho\sigma,\ \sigma(1+\rho) + cycle_{max} \cdot 2\rho]$. The explicit value is dependent on the relationship between $cycle_{max}$, $cycle_{min}$ and $Cycle$, which is determined by the pulse synchronization procedure ([5]). The explicit value of γ is presented in Section 4. This concludes the first Closure condition.

For the second Closure condition, note that Ψ_i, as defined in Section 2, represents the actual deviation of an individual correct clock (p_i) from the real-time interval during which it progresses. This is equivalent to the maximal actual difference between the clock value and real-time during a real-time interval in which real-time and the clock value were equal at the beginning of the interval. The *accuracy* of the clocks is the bound on the actual deviation of correct clocks from any finite real-time interval or rate of deviation from the progression of real-time. Thus it suffices to show that correct clocks progress with an accuracy that is a linear function of every finite real-time interval to satisfy the second Closure condition.

The clock progression has an inherent deviation from any real-time interval due to the physical clock skew. In addition, the clocks are repeatedly adjusted at every pulse in order to tighten the precision, which can further deviate the clocks progression

from the progression of the real-time during the interval. In [5] it is shown that the pulses progress with a linear envelope of any real time interval. The accuracy in a cycle equals the bound on the clock adjustment $|t_{pulse} - ET_{pulse}|$, where t_{pulse} is the clock value at the pulse at the moment prior to the adjustment of the clock to ET_{pulse}. Under perfect conditions, i.e. no clock skew and zero clock adjustment $t_{pulse} = ET_{pulse}$. This would further equal real-time should the clocks have initiated with real-time values. Thus it suffices to show that the adjustment to the clocks at every pulse is a linear function of the length of the cycle. The upper and lower bounds on the value t_{pulse} is determined by the bound on the effective cycle length and accounts for the clock skew and the accuracy of the pulses (bound on the deviation of the pulses from perfect regularity). Let $cycle_{min}$ and $cycle_{max}$ denote the lower bound and upper bound respectively on the cycle length in real-time units. Hence,

$$ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho) \leq t_{pulse} \leq ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho) \ .$$

The adjustment to the correct clocks, ADJ, is thus bounded by

$$ET_{pulse} - [ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho)] \leq 0 \leq ADJ \leq 0$$

$$\leq ET_{pulse} - [ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho)] \ ,$$

which translates to

$$ET_{prev\text{-}pulse} + Cycle - [ET_{prev\text{-}pulse} + cycle_{max} \cdot (1 + \rho)] \leq ADJ \leq$$

$$ET_{prev\text{-}pulse} + Cycle - [ET_{prev\text{-}pulse} + cycle_{min} \cdot (1 - \rho)] \ ,$$

which translates to

$$Cycle - cycle_{max} \cdot (1 + \rho) \leq ADJ \leq Cycle - cycle_{min} \cdot (1 - \rho) \ .$$

As can be seen, the bound on the adjustment to the clock is linear in the effective cycle length. The bounds on the effective cycle length are guaranteed by the pulse synchronization procedure to be linear in the default cycle length. Thus the accuracy of the clocks are within a linear envelope of any real-time interval. The actual values of $cycle_{min}$ and $cycle_{max}$ are determined by the specific pulse synchronization procedure used. This concludes the Closure condition.

□

Thus the algorithm is self-stabilizing and performs correctly with f Byzantine nodes for $n \geq 3f + 1$. □

3.2 A Clock Synchronization Algorithm without Consensus

We suggest a simple additional Byzantine self-stabilizing clock synchronization algorithm using pulse synchronization as a building block that does not use consensus.

Our second algorithm resets the clock at every pulse[9]. This approach has the advantage that the nodes never need to exchange and synchronize their clock values and thus do not need to use consensus. This version is useful for example when M, the upper-bound on the clock value, is relatively small. The algorithm has the disadvantage that for a large value of M, a large *Cycle* value is required. This enhances the effect of the clock skew, thus negatively affecting the precision and the accuracy at the end of the cycle. Note that the precision and accuracy of Cycle-Wrap-CS equals that of PBSS-Clock-Synch.

```
Algorithm Cycle-Wrap-CS
at "pulse" event                          /* received the internal pulse event */
begin
   Clock := 0;
end
```

Fig. 2. Additional CS algorithm in which the clock wraps-around every cycle

3.3 A Clock Synchronization Algorithm using an Approximate Agreement Approach

We suggest an additional self-stabilizing Byzantine clock synchronization algorithm using pulse synchronization as a building block, denoted Approx-CS.

The algorithm uses an approximate agreement approach in order to get continuous clocks with high precision and accuracy on expense of the message complexities and early-stopping property. The precision and the accuracy are $2\sigma + O(\rho)$ and thus improve on those of PBSS-Clock-Synch.

In Line 4 of Approx-CS the nodes invoke approximate-like agreement on their local clock value at the time of the last pulse, denoted Clock-at-pulse. In case that the system state was a synchronized clock_state then the resultant value $Clock_{Consensus}$ is guaranteed by the Approx_Byz_Agree to be in the range of the initial clock values of the correct nodes. If the clocks were not synchronized then the resultant agreed value may be in any range. In Line 5 every correct node sets its clock to equal the agreed clock value associated with the last pulse, $Clock_{Consensus}$, incremented with the time that has elapsed on its local timer since the pulse.

[9] This approach has been suggested by Shlomi Dolev as well.

Algorithm APPROX-CS
at "*pulse*" event /* *received the internal pulse event* */
begin
1. *Clock-at-pulse* := *Clock*;
2. Revoke possible other instances of APPROX-CS and
   clear all data structures besides *Clock-at-pulse*;
3. Wait until $\sigma(1+\rho)$ time units have elapsed since *pulse*;
4. $Clock_{Consensus}$:= APPROX_BYZ_AGREE(*Clock-at-pulse*);
5. *Clock* := ($Clock_{Consensus}$ + elapsed-time-since-pulse) mod M;

end

Fig. 3. Self-stabilizing Byzantine Approximate Clock Synchronization algorithm

Algorithm APPROX_BYZ_AGREE(value)
begin
1. Invoke BYZ_AGREEMENT() on *value*;
2. After termination of all BYZ_AGREEMENT instances (substitute missing values with 0) Do:
3. Find largest set of values within $\gamma+\sigma$ of each other (if several, choose set harboring smallest value ≥ 0);
4. Find median of the set, identify its antipode := (median + $\lfloor M/2 \rfloor$) mod M;
5. Discard the f immediate values from each side of the antipode;
6. Return the median of the remaining values;

end

Fig. 4. Self-stabilizing Byzantine Approximate Agreement

In order to be self-contained we bring the definition of *Approximate Agreement*, defined in [9].

Formally, the goal of ϵ-Approximate Agreement is to reach the following: let there be n processes $p_1, ..., p_n$, each starts with an initial value $v_i \in \mathbb{R}$ and may decide on a value $d_i \in \mathbb{R}$.

1. **(Approximate Agreement)** If p_i and p_j are correct and have decided then $|d_i - d_j| \leq \epsilon$.

2. **(Validity)** If p_i is correct and has decided then there exists two correct nodes p_j, p_k such that $v_j \leq d_i \leq v_k$, (the decision value of every correct node is in the range of the initial values of the correct nodes).

3. **(Termination)** All correct nodes eventually decide.

The approximate agreement protocol in [9] cannot be used as-is in the self-stabilization model as the notions of "highest" value and "lowest" value are not defined when nodes can initialize with values reaching their bounds, M. Faulty nodes can in this case cause different correct nodes to view the extremes of the values as complete opposites. To overcome the lack of total order relation introduced by the self-stabilization model, APPROX_BYZ_AGREE thus combines the approximate agreement algorithm of [9] with Byzantine agreement as follows: run separate Byzantine agreements in parallel on every node's value in order to agree on the value of each node. Thus all correct nodes will hold identical multisets and henceforth the heuristics of [9] will be executed on exactly the same values at all correct nodes. The APPROX_BYZ_AGREE procedure satisfies the conditions for classic approximate agreement, while being self-stabilizing.

The BYZ_AGREEMENT procedure used is the Byzantine agreement of [20], though using our BROADCAST primitive presented in Section A.2 in order to overcome the lack of any common reference to clock time among the correct nodes.

In Line 1 of APPROX_BYZ_AGREE, every node invokes Byzantine agreement on its value, within σ real-time of each other. Every instance of APPROX_BYZ_AGREE must terminate within some bounded time, thus all correct nodes can calculate a time when all the agreement instances have terminated at all correct nodes. In Line 3, after all the agreement instances have terminated and missing values are substituted with a 0, a set of supposedly synchronized values is searched for. Note that if not all instances of APPROX_BYZ_AGREE have terminated within the pre-calculated time-bound then the system must have been in a non-coherent state. Synchronized clock values can be up-to $\gamma + \sigma$ apart in the values agreed subsequent to Line2, due to the pulse uncertainty. In Line 4 the median of the set is identified, and will serve as an anchor for determining the order relation among the different values. In Line 5, the antipode (in the range $1..M$) of the median is identified; the f first values on each side of this antipode are then discarded. If the system is in a synchronized clock_state then all values that are outside of the values in the set identified earlier are discarded. Thus the median of the remaining values, returned in Line 6, is in the range of the initial values of the correct nodes.

Lemma 1. *The* APPROX_BYZ_AGREE *procedure satisfies all the conditions for ϵ-Approximate Agreement, for $\epsilon = 0$, when the system is in a synchronized clock_state*[10].

Proof. Note the validity of BYZ_AGREEMENT guarantees that the value decided by all correct nodes for node i is i's actual input value.

[10] The notion "in the range of" remains undefined if the system is not in a synchronized clock_state. Thus the validity condition remains undefined for this case.

1. **Approximate_Agreement**: All correct nodes hold the same multiset of values following all terminations of the instances of BYZ_AGREEMENT, thus they all find the same set in Line 3 and hence do the exact same operations in lines 3-5, and thus return the same value in Line 6.
2. **Validity**: Let the system be in a synchronized clock_state. Thus the agreed clock values for all correct nodes subsequent to executing Line 2 are at most $\gamma + \sigma$ apart. Hence, the largest set found in Line 3 includes at least $n - f$ values. We now seek to prove that the decision value is in the range of the initial values of the correct nodes. Since $f < n/3$ it follows that all values that are not in the range (at most f) of this set are discarded in Line 5. Thus all remaining values must be in the range of the initial values of the correct nodes. In particular, the median of the remaining values is in the range of the initial values. This completes the proof of the validity condition.
3. **Termination**: Follows from the termination of BYZ_AGREEMENT.

□

The precision γ, is the bound on the clock differences of all correct nodes at any time.

Lemma 2. *The precision of* APPROX_BYZ_AGREE *is* $2\sigma + O(\rho)$.

Proof. At the moment after all correct nodes have executed Line 5 in APPROX-CS their clocks differ by at most $\sigma + O(\rho)$, thus the clock differences are at most $\sigma + O(\rho)$ also at the forthcoming pulse invocation. The precision γ, is maximized at the moment that a correct node has set its clock subsequent to its execution of Line 5 in APPROX_BYZ_AGREE, while some other node has yet to execute this line. Following the validity condition, the agreed clock value $Clock_{Consensus}$, is within the initial clock values that was held by the correct nodes at their last pulse. As the system is in a synchronized clock_state thus these initial values were within $2\sigma + O(\rho)$ of each other. Thus the node that has just adjusted its clock, set it to a value that is within $2\sigma + O(\rho)$ of its clock at the moment before the adjustment. In particular this adjusted clock value is also within $2\sigma + O(\rho)$ of the clock value of any other correct node. This observation yields a precision of $\gamma = 2\sigma + O(\rho)$.

□

The accuracy equals the maximal clock adjustment which for the same arguments as above yields an accuracy of $2\sigma + O(\rho)$.

A self-stabilizing Byzantine approximate agreement algorithm that knows how to handle bounded, wrapping values and thus does not need to reach exact agreement on every node's value, will supposedly yield a clock synchronization algorithm with time and message complexity comparable to PBSS-CLOCK-SYNCH with precision and accuracy of APPROX-CS. To the best of our knowledge no such approximate agreement algorithm exists.

4 Analysis and Comparison to other Clock Synchronization Algorithms

Our clock synchronization algorithm PBSS-Clock-Synch requires reaching consensus in every cycle. This implies that the cycle should be long enough to allow for the consensus procedure to terminate at all correct nodes. This implies having $cycle_{min} \geq 2\sigma + 3(2f+4)d$, assuming that the Byz_Consensus procedure takes $(f+2)$ rounds of $3d$ each. The algorithm has the advantage that it uses the full time to reach consensus only following a catastrophic state in which correct nodes hold differing ET values. Once in a synchronized clock_state, all correct nodes participate in the consensus with the same initial consensus value which thus terminates within 2 communication rounds only, due to its early-stopping property. Hence, during steady state, in which the system is in a legal state, the time and message complexity overhead of PBSS-Clock-Synch is minimal.

For simplicity we also assume M to be large enough so that it takes at least a cycle for the clocks to wrap around.

Note that Ψ_i, defined in Section 2, represents the actual deviation of an individual correct clock, p_i, from a given real-time interval. The *accuracy* of the clocks is the bound on this deviation of correct clocks from any real-time interval. The clocks are repeatedly adjusted in order to minimize the accuracy. Following a synchronization of the clock values, that is targeted to occur once every *Cycle* time units, correct clocks can be adjusted by at most ADJ, where following Theorem 1,

$$Cycle - cycle_{max} \cdot (1+\rho) \leq ADJ \leq Cycle - cycle_{min} \cdot (1-\rho) \ ,$$

which, following $cycle_{min}$ and $cycle_{max}$ determined by the pulse synchronization procedure of [5] to equal $Cycle - 11d$ and $Cycle + 9d$ respectively, translates to

$$-9d(1+\rho) - \rho \cdot Cycle \leq ADJ \leq 11d(1-\rho) + \rho \cdot Cycle \ .$$

The accuracy is thus $11d + O(\rho)$ real-time units. Should the initial clock values reflect real-time then this determines the accuracy of the clocks with respect to real-time (and not only with respect to real-time progression rate), as long as the system is coherent and clocks do not wrap around.

Recall that the precision γ, is the bound on the difference between correct clock values at any time. This bound is largely determined by the maximal clock value difference at the time in which a correct node has just set its clock and some other correct node is about to do it in a short time. It is guaranteed by Theorem 1 and the pulse synchronization tightness $\sigma = 3d$ of [5], to be:

Algorithm	Self-stabilizing /Byzantine	Precision γ	Accuracy	Convergence Time	Messages
PBSS-Clock-Synch	SS+BYZ	$11d + O(\rho)$	$11d + O(\rho)$	$cycle_{max} + 3(2f+5)d$	$O(nf^2)$
Cycle-Wrap-CS	SS+BYZ	$11d + O(\rho)$	$11d + O(\rho)$	$cycle_{max}$	$O(n^2)$
Approx-CS	SS+BYZ	$3d + O(\rho)$	$3d + O(\rho)$	$cycle_{max}$	$O(nf)^2$
DHSS [8]	BYZ	$d + O(\rho)$	$(f+1)d + O(\rho)$	$2(f+1)d$	$O(n^2)$
LL-APPROX [21]	BYZ	$5\epsilon + O(\rho)$	$\epsilon + O(\rho)$	$d + O(\epsilon)$	$O(n^2)$
DW-SYNCH [12]*	SS+BYZ	0	0	$M2^{2(n-f)}$	$n^2 M 2^{2(n-f)}$
DW-BYZ-SS [12]	SS+BYZ	$4(n-f)\epsilon + O(\rho)$	$(n-f)\epsilon + O(\rho)$	$O(n)^{O(n)}$	$O(n)^{O(n)}$
PT-SYNC [18]*	SS	0	0	$4n^2$	$O(n^2)$

Table 1. Comparison of clock synchronization algorithms (ϵ is the uncertainty of the message delay). The convergence time is in pulses for the algorithms utilizing a global pulse system and in rounds for the other semi-synchronous protocols. PT-SYNC assumes the use of shared memory and thus the "message complexity" is of the "equivalent messages". The '*' denotes the use of a global pulse or global clock tick system.

$$
\begin{aligned}
\gamma &= \max[cycle_{max} \cdot (1+\rho) - Cycle + 2\rho\sigma, \\
&\qquad Cycle - cycle_{min} \cdot (1-\rho) + 2\rho\sigma,\ \sigma(1+\rho) + cycle_{max} \cdot 2\rho] \\
&= \max[9d(1+\rho) + \rho \cdot Cycle + 2\rho\sigma,\ 11d(1-\rho) + \rho \cdot Cycle + 2\rho\sigma, \\
&\qquad 3d(1+\rho) + (Cycle + 9d) \cdot 2\rho] \\
&= 11d(1-\rho) + \rho \cdot Cycle + 2\rho\sigma = 11d + O(\rho)\ .
\end{aligned}
$$

The bound on the difference between correct clock values immediately after all correct nodes have synchronized their clock value (at Line 1 or Line 5) is σ.

The only self-stabilizing Byzantine clock synchronization algorithms, to the best of our knowledge, are published in [11,12]. Two randomized self-stabilizing Byzantine clock synchronization algorithms are presented, designed for fully connected communication graphs, use message passing which allow faulty nodes to send differing values to different nodes, allow transient and permanent faults during convergence and require at least $3f+1$ processors. The clocks wrap around, where M is the upper bound on the clock values held by individual processors. The first algorithm assumes a common global pulse system and synchronizes in expected $M \cdot 2^{2(n-f)}$ global pulses. The second algorithm in [12] does not use a global pulse system and is thus partially synchronous similar to our model. The convergence time of the latter algorithm is

in expected $O((n-f)n^{6(n-f)})$ time. Both algorithms thus have drastically higher convergence times than ours.

In Table 1 we compare the parameters of our protocols to previous classic Byzantine clock synchronization algorithms, to non-Byzantine self-stabilizing clock synchronization algorithms and to the prior Byzantine self-stabilizing clock synchronization algorithms. It shows that our algorithm achieves precision, accuracy, message complexity and convergence time similar to non-stabilizing algorithms, while being self-stabilizing.

The message complexity of PBSS-Clock-Synch is solely based on the underlying Pulse and Consensus procedures. Its inherent convergence time is $cycle_{max}$. The $O(nf^2)$ message complexity as well as the $+3(2f+5)d$ additive in the convergence time come from Byz_Consensus, the specific Byzantine consensus procedure we use. The pulse synchronization procedure we use from [5] has a message complexity of $O(n^2)$ and $6 \cdot cycle$ convergence time. Note that Byz_Consensus has two early-stopping features: It stops in a number of rounds dependent on the actual number of faults and if nodes initiate with the same values (same ET values) then it stops within 2 rounds.

Note that some of the algorithms cited in Table 1 refer to ϵ, the uncertainty in message delivery, rather than d, the end-to-end communication network delay.

The DW-SYNCH and PT-SYNCH algorithms cited in Table 1 make use of global clock ticks (common physical timer). Note that this does not make the clock synchronization problem trivial as such clock ticks can not be used to invoke agreement procedures and the nodes still need to agree on the clock values. The benefit of utilizing a global pulse systems is in the optimal precision and accuracy acquired (see [12]).

References

1. E. Anceaume, I. Puaut, "*Performance Evaluation of Clock Synchronization Algorithms*", Technical report 3526,INRIA, 1998.
2. A. Arora, S. Dolev, and M.G. Gouda, "*Maintaining digital clocks in step*", Parallel Processing Letters, 1:11-18, 1991.
3. J. Brzeziński, and M. Szychowiak, "*Self-Stabilization in Distributed Systems - a Short Survey*, Foundations of Computing and Decision Sciences, Vol. 25, no. 1, 2000.
4. A. Daliot, D. Dolev and H. Parnas, "*Self-stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks*", Proc. of the 6th Symposium on Self-stabilizing Systems (SSS'03 San-Francisco), pp. 32-48, 2003.
5. A. Daliot and D. Dolev, "*Self-stabilizing Byzantine Pulse Synchronization* ", Technical Report TR2005-84, Schools of Engineering and Computer Science, The Hebrew University of Jerusalem, August 2005. A revised version appears in http://arxiv.org/abs/cs.DC/0608092 .
6. D. Dolev, J. Halpern, and H. R. Strong, "*On the Possibility and Impossibility of Achieving Clock Synchronization*", J. of Computer and Systems Science, Vol. 32:2, pp. 230-250, 1986.
7. D. Dolev, H. R. Strong, "*Polynomial Algorithms for Multiple Processor Agreement*", In Proceedings, the 14th ACM SIGACT Symposium on Theory of Computing, 401-407, May 1982. (STOC-82)

8. D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "*Dynamic Fault-Tolerant Clock Synchronization*", J. Assoc. Computing Machinery, Vol. 42, No.1, pp. 143-185, Jan. 1995.
9. D. Dolev, N. A. Lynch, E. Stark, W. E. Weihl and S. Pinter, "*Reaching Approximate Agreement in the Presence of Faults*", J. of the ACM, 33 (1986) 499-516.
10. S. Dolev, "*Possible and Impossible Self-Stabilizing Digital Clock Synchronization in General Graphs*", Journal of Real-Time Systems, no. 12(1), pp. 95-107, 1997.
11. S. Dolev, "*Self-Stabilization*", The MIT Press, 2000.
12. S. Dolev, and J. L. Welch, "*Self-Stabilizing Clock Synchronization in the presence of Byzantine faults*", Journal of the ACM, Vol. 51, Issue 5, pp. 780 - 799, 2004.
13. S. Dolev and J. L. Welch, "*Wait-free clock synchronization*", Algorithmica, 18(4):486-511, 1997.
14. M. J. Fischer, N. A. Lynch and M. Merritt, "*Easy impossibility proofs for distributed consensus problems*", Distributed Computing, Vol. 1, pp. 26-39, 1986.
15. T. Herman, "*Phase clocks for transient fault repair*", IEEE Transactions on Parallel and Distributed Systems, 11(10):1048-1057, 2000.
16. B. Liskov, "*Practical Use of Synchronized Clocks in Distributed Systems*", Proceedings of 10^{th} ACM Symposium on the Principles of Distributed Computing, 1991, pp. 1-9.
17. B. Patt-Shamir, "*A Theory of Clock Synchronization*", Doctoral thesis, MIT, Oct. 1994.
18. M. Papatriantafilou, P. Tsigas, "*On Self-Stabilizing Wait-Free Clock Synchronization*", Parallel Processing Letters, 7(3), pages 321-328, 1997.
19. F. Schneider, "*Understanding Protocols for Byzantine Clock Synchronization*", Technical Report 87-859, Dept. of Computer Science, Cornell University, Aug. 1987.
20. S. Toueg, K. J. Perry, T. K. Srikanth, "*Fast Distributed Agreement*", SIAM Journal on Computing, 16(3):445-457, June 1987.
21. J. L. Welch, and N. Lynch, "*A New Fault-Tolerant Algorithm for Clock Synchronization*", Information and Computation 77, 1-36, 1988.

A Appendix - The Consensus and Broadcast Primitives

A.1 The Byz_Consensus Procedure

The Byz_Consensus procedure can implement many of the classical Byzantine consensus algorithms. It assumes that timers of correct nodes are always within $\bar{\sigma}$ of each other. More specifically, we assume that nodes have timers that reset periodically, say at intervals $\leq cycle'$. Let $T_i(t)$ be the reading of the timer at node p_i at real time t. We thus assume that there exists a bound such that for every time t, when the system is coherent,

$$\forall i,j \text{ if } \bar{\sigma} < T_i(t), T_j(t) < cycle' - \bar{\sigma} \text{ then } |T_i(t) - T_j(t)| < \bar{\sigma} \ .$$

The bound $\bar{\sigma}$ includes all drift factors that may occur among the timers of correct nodes during that period. When the timers are reset to zero it might be that, for a short period of time, the timers may be further apart. The pulse synchronization algorithm in [5] satisfies the above assumptions and implies $\bar{\sigma} \geq d$.

The self-stabilization requirement and the deviation that may arise from any synchronization assumption imply that any consensus protocol must be carefully specified. The consensus algorithm will function properly if it is invoked when the

timers of correct nodes are within $\bar{\sigma}$ of each other. The subtle point is to make sure that an arbitrary initialization of the procedure cannot cause the nodes to block or deadlock. Below we show how to update the early stopping Byzantine Agreement algorithm of Toueg, Perry and Srikanth [20] to become self-stabilization and to make it into a general consensus (vs. agreement) procedure.

The procedure does not assume any reference to real-time and no complete synchronization of the rounds, as is assumed in [20]. Rather it resets the local timers of correct nodes at each pulse which thus makes the timers within bounds of each other. The node invokes the procedure with the value to agree on and the local timer value. In the procedure nodes also consider all messages accumulated in their buffers that were accepted prior to the invocation, if they are relevant.

We use the following notations in the description of the consensus procedure:

- Let $\bar{d}$ be the duration of time equal to $(\bar{\sigma} + d) \cdot (1 + \rho)$ time units on a correct node's timer. Intuitively, $\bar{d}$ can be assumed to be a duration of a "phase" on a correct node's timer.
- The BROADCAST primitive is the primitive defined in Section A.2 and is an adaptation of the one described in [20]. Note that an *accept* is issued within the BROADCAST primitive.

The main differences from the original protocol of [20] are:

- Instead of the General in the original protocol we use a virtual (faulty) "General" notion of a virtual node whose value is the assumed value of all correct nodes at a correct execution. It is the value with which the individual nodes invoke the procedure. Thus, every correct node does a CONSENSUS-BROADCAST of its initial Val in contrast to the original protocol in which only the General does this. If all correct nodes initiate with the same value and at the same timer time this will be the agreed value.
- The CONSENSUS-BROADCAST primitive has been modified by omitting the code dealing with the *init* messages. All correct nodes send an echo of their initial values as though they previously received the *init* message from the virtual General.
- It is assumed that the BROADCAST and CONSENSUS-BROADCAST primitives are implicitly initiated when a corresponding message arrives.

BYZ_CONSENSUS is presented in a somewhat different style. Each step has a condition attached to it, if the condition holds and the timer value assumption holds, then the step is to be executed. Notice that only the step needs to take place at a specific timer value.

```
Procedure Byz_Consensus(Val, T)                    /* invoked at p with timer T */

broadcasters := ∅; value =⊥;

Do Consensus-broadcast (General, Val, T, 1);

by time (T + 2d̄) :
   if accepted (General, v, T, 1) then
         value := v;

by time (T + (2f + 4)d̄) :
   if value ≠⊥ then
         Broadcast (p, value, T, ⌊(T_i − T)/2d⌋ + 1);
         stop and return value.

at time (T + 2rd̄) :
   if (|broadcasters| < r − 1) then
         stop and return value.

by time (T + 2rd̄) :
   if accepted (General, v', T, 1) and r − 1 distinct messages (q_i, v', T, i)
                  where ∀i, j  2 ≤ i ≤ r, and q_i ≠ q_j then
         value := v';
```

Fig. 5. The Byz_Consensus procedure

The Byz_Consensus procedure satisfies the following typical properties:

Termination: The protocol terminates in a finite time;
Agreement: The protocol returns the same value at all correct nodes;
Validity: If all correct nodes invoke the protocol with the same value and time, then the protocol returns that value;

It also satisfies the following **early stopping** properties:

ES-1 If all correct nodes invoke the protocol with the same consensus value and with the same timer value, then they all stop within two "rounds" of information exchange among correct nodes.
ES-2 If the actual number of faults is $f' \leq f$ then the algorithm terminates by $\min[T + (2f' + 6)\bar{d},\ T + (2f + 4)\bar{d}]$ on the timer of each correct node.

Notice that [ES-1] takes in practice significantly less time than the specified upper bound on the message delivery time.

We first prove the properties of the Consensus-broadcast primitive and later we prove the correctness of the Byz_Consensus procedure.

The Consensus-broadcast primitive and the Broadcast primitive (defined in

```
Procedure Consensus-broadcast (General, v, τ, 1)
                                  /* invoking a broadcast simulating the General */
                          /* nodes send specific message with the same τ only once */
                        /* multiple messages sent by an individual node are ignored*/

send (echo, General, v, τ, 1) to all;

by time (τ + d̄) :
   if received (echo, General, v, τ, 1) from ≥ n − 2f distinct nodes then
         broadcasters := broadcasters ⋃ {General} ;
   if received (echo, General, v, τ, 1) from ≥ n − f distinct nodes q then
         send (echo', General, v, τ, 1) to all;

at any time:
   if received (echo', General, v, τ, 1) from ≥ n − 2f distinct nodes then
         send (echo', General, v, τ, 1) to all;
   if received (echo', General, v, τ, 1) from ≥ n − f distinct nodes then
         accept (General, v, τ, 1);
```

Fig. 6. Consensus-broadcast

Section A.2) satisfy the following [TPS-*] properties of Toueg, Perry and Srikanth [20], which are phrased in our system model.

TPS-1 *(Correctness)* If a correct node p does Broadcast (p, m, τ, k) by $\tau + (2k-2)\bar{d}$ on its timer, then every correct node accepts (p, m, τ, k) by $\tau + 2k\bar{d}$ on its timer.

TPS-2 *(Unforgeability)* If no correct node p does a Broadcast (p, m, τ, k), then no correct node accepts (p, m, τ, k).

TPS-3 *(Relay)* If a correct node accepts (p, m, τ, k) by $\tau + 2r\bar{d}$, for $r \geq k$, on its timer then every other correct node accepts (p, m, τ, k) by $\tau + (2r+2)\bar{d}$ on its timer.

TPS-4 *(Detection of broadcasters)* If a correct node accepts (p, m, τ, k) by $\tau + 2r\bar{d}$, on its timer then every correct node has $p \in broadcasters$ by $\tau + (2k+1)\bar{d}$ on its timer. Furthermore, if a correct node p does not Broadcast any message, then a correct node can never have $p \in broadcasters$.

Additionally, the Consensus-broadcast primitive also satisfies:

TPS-5 *(Uniqueness)* If a correct node accepts $(General, m, \tau, 1)$, then no correct node ever accepts $(General, m', \tau, 1)$ with $m' \neq m$.

Notice the differences from the original properties. The detection property does not require having $r \geq k$. In general, the relay property holds even earlier than $r \geq k$. The condition $r \geq k$ of when the property can be guaranteed is used to simplify the possible cases. At $r < k$, if an accept takes place as a result of getting $n - f$ echo messages, the adversary may cause the relay to take $3\bar{d}$ by rushing messages to one correct node and delay messages to and from others.

Theorem 2. *The* CONSENSUS-BROADCAST *primitive satisfies the five [TPS-*] properties.*

Proof.
Correctness: If all correct nodes send $(echo, General, v, \tau, 1)$ at time τ on their timers, then by Lemma 5 every correct node accepts $(General, v, \tau, 1)$ from $n - f$ correct nodes by $\tau + \bar{d}$ on its timer. Thus each correct node sends $(echo, General, v, \tau, 1)$ by that time and will accept $(General, v, \tau, 1)$ by $\tau + 2\bar{d}$ on their timers.

Unforgeability: If all correct nodes hold the same initial value v then no correct node will send $(echo, General, v', 1)$, thus no correct node will receive $n - f$ distinct $(echo, General, v', 1)$ messages. Therefore, no correct node will send $(echo', General, v', 1)$, and no correct node will ever receive $n - 2f$ or $n - f$ distinct $(echo', General, v', 1)$ messages. Thus, no correct node can accept $(General, v', 1)$.

Relay: If a correct node accepts $(General, v, \tau, 1)$ by $\tau + 2r\bar{d}$ on its timer, then it received $n - f$ distinct $(echo', General, v, \tau, 1)$ message by that time. $n - 2f$ of these were sent by correct nodes and by Lemma 5 all of them will reach all correct nodes by $\tau + (2r+1)\bar{d}$. As a result, all such correct nodes will send $(echo', General, v, \tau, 1)$, which will be received by all correct nodes. Hence, by $\tau + (2r + 2)\bar{d}$ on their timers, all correct nodes will hold $n - f$ distinct $(echo', General, v, \tau, 1)$ messages and will thus accept $(General, v, \tau, 1)$.

Detection of broadcasters: If a correct node q' accepts $(General, v, \tau, 1)$ by time $\tau + 2r\bar{d}$ on its timer, then node q' should have received at least $n - f$ distinct $(echo', General, v, \tau, 1)$ messages, at least $n - 2f$ of which are from correct nodes. Let q be the first correct node to ever send $(echo', General, v, \tau, 1)$. If q sent it as a result of receiving $n - f$ such messages, then q is not the first to send. Therefore, it should have sent it as a result of receiving $n - f$ $(echo, General, v, \tau, 1)$ messages by time $\tau + \bar{d}$. Thus, at least $n - 2f$ such messages were sent by correct nodes by time τ on their timers and would arrive at all correct nodes by time $\tau + \bar{d}$ on their timers. As a result, all will have $General \in broadcasters$.

Uniqueness: Notice that if a correct node sends $(echo', General, v, \tau, 1)$ by time $\tau + \bar{d}$, then no correct node sends $(echo', General, v', 1)$ at any later time. Otherwise, similarly to the arguments in proving the previous property we get that at least $n - f$ nodes sent $(echo, General, v, \tau, 1)$ and $n - f$ nodes sent $(echo, General, v', 1)$. Since $n > 3f$, this implies that at least one correct node sent both $(echo, General, v, \tau, 1)$ and $(echo, General, v', 1)$, and this is not allowed.

Also note that if a correct node accepts $(General, v, \tau, 1)$, then at least one correct node sends $(echo', General, v, \tau, 1)$, which yields the proof of the *Uniqueness* property. □

Nodes stop participating in Byz_Consensus when they are instructed to do so. They stop participating in the Broadcast primitive $2\bar{d}$ after they terminate Byz_Consensus.

Definition 7.
A node **returned** *a value m if it has stopped and returned value $= m$.*
A node p **decides** *if it stops at that timer time and returns a value $\neq \perp$.*
A node p **aborts** *if it stops and returns $\perp$.*

Theorem 3. *The* Byz_Consensus *procedure satisfies the Termination property. When $n > 3f$, it also satisfies Agreement, Validity and the two early stopping conditions.*

Proof. We prove the five properties of the theorem. We build up the proof through the following arguments.

Lemma 3. *If a correct node aborts at time $T + 2r\bar{d}$ on its timer, then no correct node decides at a time $T + 2r'\bar{d} \geq T + 2r\bar{d}$ on its timer.*

Proof. Let p be a correct node that aborts at time $T+2r\bar{d}$. In this case it should have identified exactly $r-2$ broadcasters by that time. By the detection of broadcasters property [TPS-4] no correct node will ever accept $(General, v, T, 1)$ and $r-2$ distinct messages (q_i, v, T, i) for $2 \leq i \leq r-1$, since that would have caused all correct nodes to hold $r-1$ broadcasters by time $T + (2r-1)\bar{d}$ on their timers. Thus, no correct node can decide at local-time $T + 2r'\bar{d} \geq T + 2r\bar{d}$. □

Lemma 4. *If a correct node decides by time $T+2r\bar{d}$ on its timer, then every correct node decides by time $T + 2(r+1)\bar{d}$ on its timer.*

Proof. Let p be a correct node that decides by time $T+2r\bar{d}$ on its timer. We consider the following cases:

1. $r = 1$: No correct node can abort by time $T + 2\bar{d}$, since the inequality will not hold. Node p must have accepted $(General, v, T, 1)$ by $T + 2\bar{d}$. By the relay property [TPS-3] all correct nodes will accept $(General, v, T, 1)$ by $T+4\bar{d}$ on their timers. Moreover, p invokes Broadcast $(p, v, T, 2)$, by which the correctness property [TPS-1] will be accepted by all correct nodes by time $T + 4\bar{d}$ on their timers. Thus, all correct nodes will have *value* $\neq \perp$ and will Broadcast and stop by time $T + 4\bar{d}$ on their timers.

2. $2 \leq r \leq f+1$. Node p must have accepted $(General, v, T, 1)$ and also accepted $r-1$ distinct (q_i, v, T, i) messages for all $i, 2 \leq i \leq r$, by time $T+2r\bar{d}$ on its timer. By Lemma 3, no correct aborts by that time. By Relay property [TPS-3] each (q_i, v, T, i) message will be accepted by all correct nodes by time $T + (2r+2)\bar{d}$ on their timers. Node p does BROADCAST $(p, v, T, r+1)$ before stopping. By the correctness property, this message will be accepted by all correct nodes by time $T + (2r+2)\bar{d}$ on their timers. Thus, no correct node will abort by $T + (2r+2)\bar{d}$ and all correct nodes will have $value \neq \perp$ and will decide and stop by that time.
3. $r = f+2$. Node p must have accepted (q_i, v, T, i) messages for all $i, 2 \leq i \leq f+2$, by $T + (2f+4)\bar{d}$ on its timer, where the $f+1$ q_i's are distinct. At least one of these $f+1$ nodes, say q_j, must be correct. By the Unforgeability property [TPS-2] q_j, invoked BROADCAST (q_j, v, T, j) by time $T + (2j)\bar{d}$ on its timer, and decided. Since $j \leq f+1$ the above arguments imply that by $T + (2f+4)\bar{d}$ on their timers all correct will decide.

□

Lemma 4 implies that if a correct node decides at time $T + 2r\bar{d}$ on its timer, then no correct node aborts at round $T + 2r'\bar{d}$. Lemma 3 implies the other direction.

Termination: Lemma 4 implies that if any correct node decides, all decide and stop. Assume that no correct node decides. In this case, no correct node ever invokes a BROADCAST $(q, v, T, \_)$. By detection of broadcasters property [TPS-4], no correct node will ever be considered as broadcaster. Therefore, by time $T + ((2f+4)\bar{d}$ on their timers, all correct nodes will have at most f broadcasters and will abort and stop. □

Agreement: If no correct node decides, then all abort, and return to the same value. Otherwise, let p be the first correct node to decide. Therefore, no correct node aborts. The value returned by p is the value v of the accepted $(General, v, 1)$ message. By Properties [TPS-3] and [TPS-5] all correct nodes accept $(General, v, T, 1)$ and no correct node accepts $(General, v', T, 1)$ for $v \neq v'$. Thus all correct nodes return the same value. □

Validity: Let all the correct nodes begin with the same value v' and invoke the protocol with the same timer time (T). Then, by time $T + \bar{d}$ on their timers, all correct nodes receive at least $n - 2f$ distinct $(echo, General, v', T, 1)$ messages via the CONSENSUS-BROADCAST primitive and send $(echo', General, v', T, 1)$ messages to all. Hence, all nodes receive at least $n - f$ distinct $(echo', General, v', T, 1)$ messages by $T + 2\bar{d}$ on their timers and thus accept $(General, v', T, 1)$. Hence in the

BYZ_CONSENSUS procedure all correct nodes set their value to v'. By $T + 2\bar{d}$ on their timers, all correct nodes will stop and return v'. □

Early-stopping: The first early stopping property [ES-1] is directly implied from the proof of the validity property. Correct nodes proceed once they receive messages from $n - f$ nodes, thus it is enough to receive messages from all correct nodes. The proof of the second early stopping property [ES-2] is identical to the proof of the termination property. By time $T + (2f' + 4)\bar{d}$ all will abort unless any correct node invokes BROADCAST by that time on its timer. This implies that by $T + (2f' + 6)\bar{d}$ on their timers all correct nodes will always terminate, if the actual number of faults f' is less than f. □

Thus the proof of the theorem is concluded. □

A.2 The BROADCAST Primitive

This section presents the **Broadcast** (and *accept*) primitive that is used by the BYZ_CONSENSUS procedure presented earlier, in Section A.1. The primitive follows the primitive of of Toueg, Perry, and Srikanth [20], though here it is presented in a real-time model.

In the original synchronous model, nodes advance according to phases. This intuitive lock-step process clarifies the presentation and simplifies the proofs. In this section, the discussion carefully considers the various time consideration and proves that nodes can rush through the protocol and do not to need to wait for a completion of a "phase" in order to move to the next step of the protocol.

Note that when a node invokes the procedure it evaluates all the messages in its buffer that are relevant to the procedure.

The BROADCAST primitive satisfies the four [TPS-*] properties, under the assumption that $n > 3f$. The proofs below follow closely to the original proofs of [20], in order to make it easier for readers that are familiar with the original proofs.

Lemma 5. *If a correct node p_i sends a message at timer time $T_i \leq \tau + r\bar{d}$ on p_i's timer it will be recieved by each correct node p_j by timer time $\tau + (r + 1)\bar{d}$ on p_j's timer.*

Proof. Assume that node p_i sends a message at real time t with timer time $T_i(t) \leq \tau + r\bar{d}$. Thus, $T_i(t) \leq \tau + r(\bar{\sigma} + d)(1 + \rho)$. It should arrive at every correct timer p_j within $d(1 + \rho)$ on any correct node's timer. Recall that $|T_i(t) - T_j(t)| < \bar{\sigma}(1 + \rho)$. If $T_j \geq T_i$ we are done. Otherwise,

$$T_j(t) \leq T_i(t) + \bar{\sigma}(1 + \rho) \leq \tau + r(\bar{\sigma} + d)(1 + \rho) + \bar{\sigma}(1 + \rho) .$$

```
Procedure BROADCAST (p, m, τ, k)
                                        /* executed per such quadruple */
                              /* nodes send specific message with the same τ only once */
                        /* multiple messages sent by an individual node are ignored */

node p sends (init, p, m, τ, k) to all nodes;

by time (τ + (2k − 1)d̄) :
   if (received (init, p, m, τ, k) from p then
         send (echo, p, m, τ, k) to all;

by time (τ + 2kd̄) :
   if (received (echo, p, m, τ, k) from ≥ n − 2f distinct nodes q then
         send (init', p, m, τ, k) to all;
   if (received (echo, p, m, τ, k) msgs from ≥ n − f distinct nodes then
         accept (p, m, τ, k);

by time (τ + (2k + 1)d̄) :
   if (received (init', p, m, τ, k) from ≥ n − 2f then
         broadcasters := broadcasters ⋃ {p};
   if (received (init', p, m, τ, k) from ≥ n − f distinct nodes then
         send (echo', p, m, τ, k) to all;

at any time:
   if (received (echo', p, m, τ, k) from ≥ n − 2f distinct nodes then
         send (echo', p, m, τ, k) to all;
   if (received (echo', p, m, τ, k) from ≥ n − f distinct nodes) then
         accept (p, m, τ, k);
end
```

Fig. 7. BROADCAST primitive

By the time (say t') that the message arrives to p_j we get

$$T_j(t') \leq \tau + r(\bar{\sigma} + d)(1 + \rho) + \bar{\sigma}(1 + \rho) + d(1 + \rho) \leq \tau + (r + 1)\bar{d}\,.$$

□

Lemma 6. *If a correct node ever sends* $(echo', p, m, \tau, k)$ *then at least one correct node must have sent* $(echo', p, m, \tau, k)$ *by timer time* $\tau + (2k + 1)\bar{d}$.

Proof. Let t be the earliest timer time by which any correct node q sends the message $(echo', p, m, \tau, k)$. If $t > \tau + (2k + 1)\bar{d}$, node q should have received $(echo', p, m, \tau, k)$ from $n - 2f$ distinct nodes, at least one of which from a correct node that was sent prior to timer time $\tau + (2k + 1)\bar{d}$. □

Lemma 7. *If a correct node ever sends* $(echo', p, m, \tau, k)$ *then p's* $(init, p, m, \tau, k)$ *must have been received by at least one correct node by time* $\tau + (2k - 1)\bar{d}$.

Proof. By Lemma 6, if a correct node ever sends $(echo', p, m, \tau, k)$, then some correct node q should send it by time timer $\tau + (2k+1)\bar{d}$. By the procedure, q have received $(init', p, m, \tau, k)$ from at least $n - f$ nodes by timer time $\tau + (2k+1)\bar{d}$. At least one of them is correct who have received $n - 2f$ $(echo, p, m, \tau, k)$ by timer time $\tau + 2k\bar{d}$. One of which was sent by correct node that should have received $(init, p, m, \tau, k)$ before sending $(echo, p, m, \tau, k)$ by timer time $\tau + (2k-1)\bar{d}$. □

Theorem 4. *The* BROADCAST *primitive presented in Figure 7 satisfies properties [TPS-1] through [TPS-4].*

Proof.

Correctness: Assume that a correct node p sends (p, m, τ, k) by $\tau + (2k-2)\bar{d}$ on its timer. Every correct node receives $(init, p, m, \tau, k)$ and sends $(echo, p, m, \tau, k)$ by $\tau + (2k-1)\bar{d}$ on its timer. Thus, every correct node receives $n - f$ $(echo, p, m, \tau, k)$ from distinct nodes by $\tau + (2k-1)\bar{d}$ on its timer and accepts (p, m, τ, k).

Unforgeability: If no correct node p does a BROADCAST (p, m, τ, k), it does not send $(init, p, m, \tau, k)$, and no correct node will send $(echo, p, m, \tau, k)$ by $\tau + (2k-1)\bar{d}$ on its timer. Thus, no correct node accepts (p, m, τ, k) by $\tau + 2k\bar{d}$ on its timer. If a correct node would have accepted (p, m, τ, k) at a later time it can be only as a result of receiving $n - f$ $(echo', p, m, \tau, k)$ distinct messages, some of which must be from correct nodes. By Lemma 7, p should have sent $(init, p, m, \tau, k)$, a contradiction.

Relay: Notice that $r \geq k$, thus even if nodes issue an accept at earlier time, the claim holds for the specified times.

The subtle point is when a correct node issues an accept as a result of getting echo messages. If $r = k$ and the correct node, say q, have received $(echo, p, m, \tau, k)$ from $n - f$ nodes by $\tau + 2k\bar{d}$ on its timer. At least $n - 2f$ of them were sent by correct nodes. Since every correct node among these has sent its message by $\tau + (2k-1)\bar{d}$, all those messages should have arrived to every correct node by $\tau + 2k\bar{d}$ on its timer. Thus, every correct node should have sent $(init', p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. As a result, every correct node will receive $n - f$ such messages by $\tau + (2k+1)\bar{d}$ on its timer and will send $(echo', p, m, \tau, k)$ by that time, which will lead all correct nodes to accept (p, m, τ, k) by $\tau + (2r+2)\bar{d}$ on its timer.

Otherwise, the correct node, say q, accepts (p, m, τ, k) by $\tau + 2r\bar{d}$ on its timer as a result of receiving $n - f$ $(echo', p, m, \tau, k)$ by that time. Since $n - f$ of these are from correct nodes, they should arrive at any correct node by $\tau + (2r+1)\bar{d}$ on their timers. As a result, by $\tau + (2r+1)\bar{d}$, all correct nodes would send $(echo', p, m, \tau, k)$ and by $\tau + (2r+2)\bar{d}$ on their timers all will accept (p, m, τ, k).

Detection of broadcasters: As in the original proof, we first argue the second part. Assume that a correct node q adds node p to *broadcasters*. It should have received $n - 2f$ $(init', p, m, \tau, k)$ messages. Thus, at least one correct node has sent $(init', p, m, \tau, k)$ as a result of receiving $n - 2f$ $(echo, p, m, \tau, k)$ messages. One of these

should be from a correct node that has received the original BROADCAST message of p.

To prove the first part, we consider two similar cases to support the Relay property. If $r = k$ and the correct node, say q, accepts (p, m, τ, k) as a result of receiving $n - f$ $(echo, p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. At least $n - 2f$ of them were sent by correct nodes. Since every correct node among these has sent its message by $\tau + (2k - 1)\bar{d}$, all those messages should have arrived at every correct node by $\tau + 2k\bar{d}$ on its timer. Thus, every correct node should have sent $(init', p, m, \tau, k)$ by $\tau + 2k\bar{d}$ on its timer. Consequently, all correct nodes will receive $n - f$ such messages by time $\tau + (2k + 1)\bar{d}$ and will add p to $broadcasters$.

Otherwise, q accepts (p, m, τ, k) as a result of receiving $(echo', p, m, \tau, k)$ from $n - f$ nodes by $\tau + 2r\bar{d}$ (for $r \geq k$) on its timer. By Lemma 6 a correct node sent $(echo', p, m, \tau, k)$ by $\tau + (2k + 1)\bar{d}$. It should have received $n - f$ $(init', p, m, \tau, k)$ messages by that time. All such messages that were sent by correct nodes were sent by $\tau + 2k\bar{d}$ on their timers and should arrive at every correct node by $\tau + (2k+1)\bar{d}$ on its timer. Since there are at least $n - 2f$ such messages, all will add p to $broadcasters$ by $\tau + (2k + 1)\bar{d}$ on their timers. □